Appendix A

The value of an object of built-in type that is not explicitly initialized depends on where it is defined. Non-local variables, defined outside any function body, are zero initialized[1]. With one exception, local static objects (we may cover later in class). Local variables, defined inside a function, are default initialized[2]. The value of an default initialized variable of built-in type is undefined.

Caution: Uninitialized Variables Cause Run-Time Problems

An uninitialized variable has an indeterminate value. Trying to use the value of an uninitialized variable has no error in syntax. The compiler will not catch the error for you. It is a logical error that is often hard to debug.

It is recommended to initialize every object of built-in type. Although it is not always necessary, it is easier and safer to provide an initializer until you can be certain it is safe to omit the initializer.

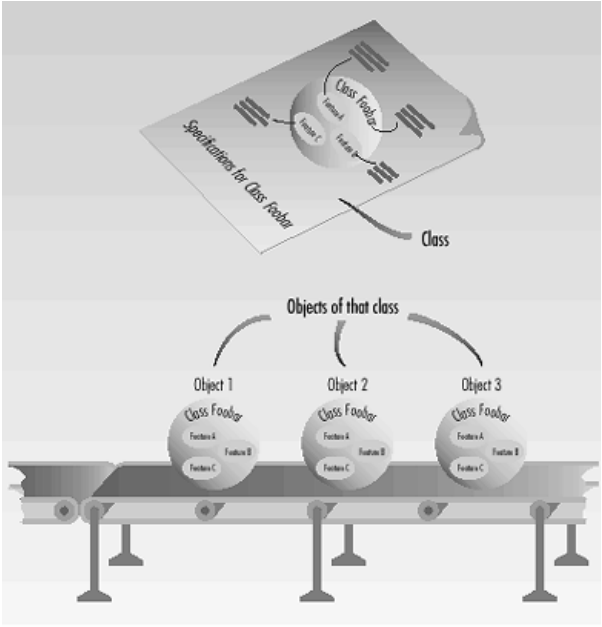
Chapter 7: Classes (A First Look)

Overview

We use **classes** to define **our own data types**. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier understand, and even easier to write, debug, and modify.

[1] https://en.cppreference.com/w/cpp/language/initialization

[2] https://en.cppreference.com/w/cpp/language/default_initialization



Previously, `struct` was introduced. It is the same as a class, but having all members public by default.

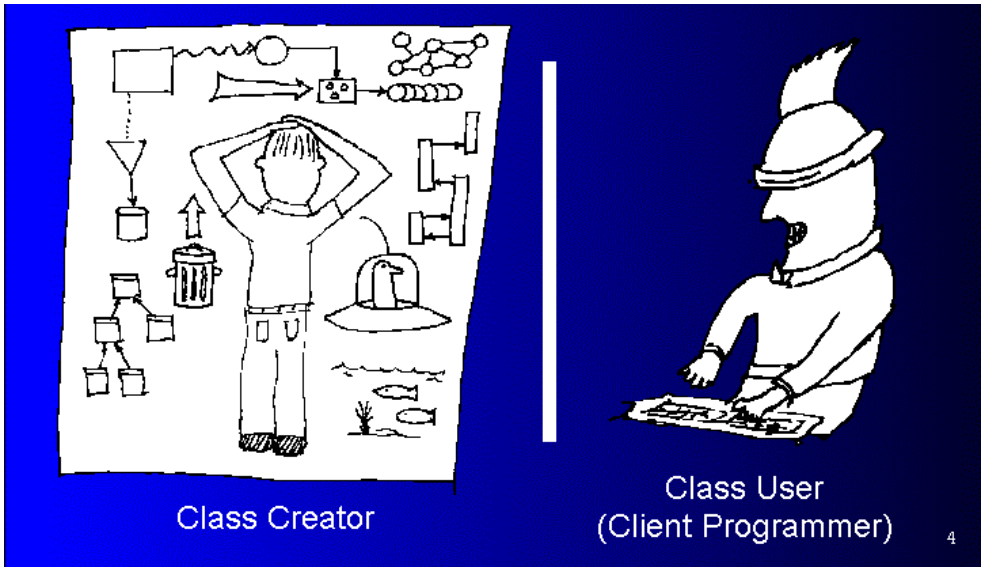
Abstract Data Types (ADT)

The fundamental ideas behind classes are data abstraction and encapsulation. **Data abstraction** means representation of information in terms of its interfaces (member or non-member functions, e.g., `push_back` in `vector`) with the user. **Encapsulation** means hiding details of implementation (e.g., internal memory management in `vector`) from the user.

A class that uses data abstraction and encapsulation is an **abstract data type**. ADT is a type (a type is the same as a class) for objects whose behavior is defined by a set of values and a set of operations.

In an ADT, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. **They can instead think abstractly about what the type does**.

Different roles in programming.



Is the `Student` data type (listed below) an **abstract** data type (ADT)? `Student` is not an ADT. An ADT must have well-defined interfaces (member or non-member functions).

```
struct Student{
    std::string name;
    int id;
    int age;
};
```

It lets users access its data members and forces users to write their own operations (functions). To make `Student` an abstract type, we need to define operations for users of `Student` to use. Once `Student` defines its own operations, we can encapsulate (that is, hide) its data members.

7.1 Defining Abstract Data Types

7.1.1 Designing the `Sales_data` Class

The following are some details on the `Sales_data` class:

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

We will add operations for data abstraction. The interface to `Sales_data` consists of the following operations:

- An `isbn` **member function** to return the object's ISBN
- A `combine` **member function** to add one `Sales_data` object into another
- An `add` function to add two `Sales_data` objects
- A `read` function to read data from an `istream` into a `Sales_data` object
- A `print` function to print the value of a `Sales_data` object on an `ostream`.

7.1.2 Adding Member Functions to the `Sales_data` Class

We define and declare **member functions** similarly to ordinary functions. Member functions **must be declared** inside the class. Member functions may be **defined** inside the class itself or outside the class body.

We can add a member function to the `Sales_data` class:

```
struct Sales_data {
    // declare a member function
    Sales_data& combine(const Sales_data&);

    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Nonmember functions that are part of the interface, such as `add`, `read`, and `print`, are **declared and defined** outside the class.

```
struct Sales_data {
    // add member function
    Sales_data& combine(const Sales_data&);

    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream& print(std::ostream&, const Sales_data&);
std::istream& read(std::istream&, Sales_data&);
```

Constant member functions[3]

A function becomes `const` when the `const` keyword is used in the function's declaration. The idea of `const` functions is not to allow them to modify the object on which they are called.

```
#include <iostream>
using namespace std;

class Test {
    int value;

public:
    Test(int v = 0) { value = v; }

    int getValue() const { return value; }
};

int main(){
    Test t(20);
    cout << t.getValue();
    return 0;
}
```

It is recommended to make functions `const` whenever possible to avoid accidental changes to objects.

Defining Member Functions

Member functions may be **defined** inside the class itself or outside the class body. In general, if the function body consists of more than three lines, we should do it outside the class.

```
struct Sales_data {
    // member functions
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;

    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

[3] https://www.geeksforgeeks.org/const-member-functions-c/

```
// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream& print(std::ostream&, const Sales_data&);
std::istream& read(std::istream&, Sales_data&);
```

(1) Declare and define member function inside the class definition

Sales_data.h

```
struct Sales_data {
    std::string isbn() const { return bookNo; }
    ...
```

(2) Declare member function inside the class definition and define it outside

Sales_data.h

```
struct Sales_data {
    ...
    double avg_price() const;
```

Sales_data.cpp

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

The **::** symbol is called the **scope resolution operator**. It is needed to indicate that these are class member functions and to tell the compiler which class they belong to.

The `Sales_data` member function returns the average price at which the books were sold. This function is not intended for general use. It will be part of the implementation, not part of the interface. We can use access control (private) to avoid user using this function. We'll talk about this later.

The following is just an example of using a class for programming.

Person.h

```
#ifndef PERSON_H
#define PERSON_H

#include <string>

struct Person {
```

```
    void setName(const std::string& s){name = s;}
    void setAge(unsigned num){age = num;}
    std::string getName() const {return name;} // later note
    unsigned getAge() const {return age;}

    std::string name;
    unsigned age;
};

#endif
```

PersonClient.cpp

```
#include <iostream>
#include "Person.h"

using namespace std;

int main(){
    Person p;
    p.setName("John");
    p.setAge(25);
    cout << "Name: "<< p.getName()<<" age: "<<p.getAge()<< endl;
}
```

The `this` Pointer

A special pointer called `this` is implicitly defined to point to the object used to invoke a member function. (Basically, `this` is passed as a hidden argument to the method.)

Let us now look at the `combine` member function to illustrate the concept. The `combine` member function is intended to act like the compound assignment operator, +=.

```
total += trans; // update the running total
```

The object on which this member function is called represents the left-hand operand, the object `total`. The right-hand operand `trans` is passed as an explicit argument:

```
total.combine(trans); // update the running total
```

The `this` pointer is very useful when we need to return **the object** on which the member function was called.

```
Sales_data& Sales_data::combine(const Sales_data &rhs){
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return ???;
}
```

We will need to return **the object** on which the function was called. We use the `this` pointer.

The function call `total.combine(trans)` sets `this` to the address of the `total` object and makes `this` available to the `combine` method.

Q: Now complete **`???`** in the definition of `combine` member function.

A:

```
    return *this;
```

To return the object itself, you need to dereference the pointer: `*this`.

The following syntax provides better clarity for the function definition.

```
Sales_data& Sales_data::combine(const Sales_data &rhs){
    this->units_sold += rhs.units_sold;
    this->revenue += rhs.revenue;
    return *this;
}
```

The `this->units_sold` is the same as `(*this).units_sold`. The `->` operator is used to access members when using a pointer.

Chained operations

If we would like to concatenate a sequence of `set` actions into a single expression such as:

```
    Person p;
    p.setName("John").setAge(25);
    p.setAge(25).setName("John");
```

This is called chained operations. To accomplish this, we can modify implementations of `setAge` and `setName` in the `Person` class from

```
void setName(const std::string& s){name = s;}
void setAge(unsigned num){age = num;}
```

to

```
Person& setName(const std::string& s){name = s; return *this;}
Person& setAge(unsigned num){age = num; return *this;}
```

const Member Functions and this

```
struct Sales_data {
// new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    ...
```

The const relates to this. The this is a pointer to const so a const member function cannot change the object on which it is called. It implies that that avg_price and isbn may read but not modify/write to the data members of the objects on which they are called.

7.1.3 Defining Non-member Class-Related Functions

Class authors often define **nonmember functions**, such as the non-member add, read, and print functions. Although such functions define operations that are conceptually part of the interface of the class, they are not part of the class itself.

Nonmember functions that are conceptually part of a class, but not defined inside the class, are typically declared (but not defined) in **the same header** as the class itself. Then users need to include only one file to use any part of the interface.

Sales_data.h

```
struct Sales_data {
    ...
};

// nonmember Sales_data interface function
std::ostream &print(std::ostream&, const Sales_data&);
```

Sales_data.cpp

```
ostream &print(ostream &os, const Sales_data &item){
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

Client code

```
print(cout, total) << endl; // print the results
```

Notice that the IO classes are types that **cannot be copied**, so we may only pass them by reference. Moreover, reading or writing to a stream changes that stream, so both functions take ordinary references, not references to `const`.

In-class member initializers

This is also called the **default member initializers**[4]. We can initialize data members of a class with default values by the **in-class member initializers** as in the following:

```
class A {
    public:
        int a = 7;
};
```

If a member is initialized by both an in-class initializer and a constructor (which will be shown later in this lecture), only the constructor's initialization is done (it "overrides" the default).

7.1.4 Constructors

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more special member functions known as **constructors**.

In other words, data members of the class are initialized through a constructor. A constructor is a special member function that is called whenever an object of the class is created.

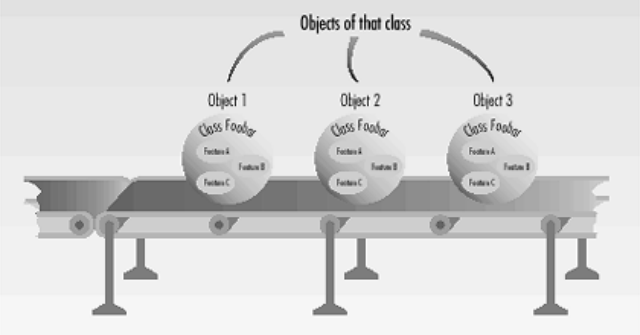
In C++, constructors are member functions with **the same name as the class**; they have **no return type**. Like other member functions they take a (possibly empty) parameter list and have a function body.

[4] For more information on default member initializers, see URL: https://abseil.io/tips/61

A class can have multiple constructors. Each constructor must differ from the others via the parameters, having an **overloading** relationship.

The constructor's parameters specify the initializers that may be used when creating objects of the class type. These initializers are used to initialize the data members of the newly created object. Constructors should ensure that every data member is initialized.

```
string s("hello");  // constructor: takes string literal
string s;           // default constructor: empty string
```



The **default constructor** is the one that takes no arguments. The default constructor says what happens when we define an object but do not supply an (explicit) initializer:

```
vector<int> vi;  // default constructor: empty vector
string s;  // default constructor: empty string
Sales_data total;   // default constructor: initialize ???
```

Synthesized Default Constructor

The default constructor is special and if our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us.

This compiler-generated default constructor is known as the **synthesized default constructor**. It initializes each data members: of the class as follows:

- If there is an in-class member initializer, use it to initialize the member.
- Otherwise, default-initialize[5] the member.

[5]Default initialization is performed when a variable is constructed with no initializer. For more information on default initialization, please take a look at URL: https://en.cppreference.com/w/cpp/language/default_initialization

For our `Sales_data` class we'll define **three** constructors with the following parameters:

- A `const string&` representing an ISBN, an `unsigned` representing the count of how many books were sold, and a `double` representing the price at which the books sold.
- A `const string&` representing an ISBN. This constructor will use default values for the other members.
- An empty parameter list (i.e., the default constructor). We must define the default constructor by ourselves. The compiler will not generate the **synthesized default constructor** because we have defined other constructors.

```
struct Sales_data {
    // constructors added
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
...
```

What `= default` Means

```
Sales_data() = default;
```

We can ask the compiler to generate the default constructor by writing `= default`.

Constructor Initializer List

```
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
```

The colon up to the open curly is the **constructor initializer list**. It specifies initial values for one or more data members of the object being created.

The constructor initializer is a list of member names, each of which is followed by that member's initial value in parentheses (or inside curly braces {}).

democlass.h

```
#ifndef DEMOCLASS_H
#define DEMOCLASS_H

struct DemoClass {
    // constructor
    Democlass(int a=0, int b=1): itemA(a), itemB(b){}
```

```
    int itemA, itemB;
};
#endif
```

In the above, we have a constructor using an initializer list and default argument. It (1) initialize `itemA` with the value of `0` and `itemB` with the value of `1` and (2) in the initializer list, initialize `itemA` with the first parameter and `itemB` with the second parameter.

Illustrative example

CircleClient.cpp

Here is a simple `Circle` class (`Circle.h` and `Circle.cpp`) so one can set its radius through a constructor or by a member function. In addition, the `Circle` object can report its radius and area when we print the object. Below are a sample client code and output:

```
#include <iostream>
#include "Circle.h"

using namespace std;

int main(){
   Circle c1;
   print (cout, c1);
   c1.setRadius(1);    // This sets c1 radius to 1
   print (cout, c1);
   Circle c2(2.5);     // This sets c2 radius to 2.5
   print (cout, c2);
   return 0;
}
```

```
Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349
```

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle{
    Circle() = default;
    Circle(double r):radius{r}{}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
    double radius = 0.0;
    const double PI = 3.14159;
};
```

```
std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c){
    os << "Circle radius: " << c.radius << " area: "
       << c.getArea() << endl;
    return os;
}
```

Using the constructor initializer list v.s. assignment in function body

For performance reasons, the constructor initializer list is preferred over assignment in the constructor's function body[6].

```
// Without Initializer List
class MyClass {
    Type variable;
public:
    MyClass(Type a) {  // Assume that Type is an already
                       // declared class and it has appropriate
                       // constructors and operators
      variable = a;
    }
};
```

Here compiler follows following steps to create an object of type MyClass

1. Type's constructor is called first for "a".
2. Default construct "variable"
3. The assignment operator of "Type" is called inside body of MyClass() constructor to assign

```
    variable = a;
```

4. And then finally destructor of "Type" is called for "a" since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
class MyClass {
    Type variable;
public:
```

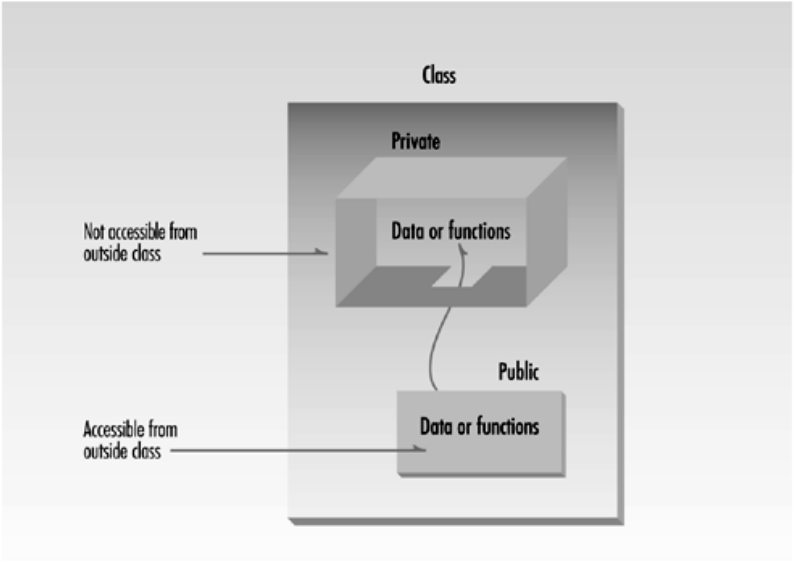
[6] https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/

```
    MyClass(Type a):variable(a) {
                        // Assume that Type is an already
                       // declared class and it has appropriate
                       // constructors and operators
    }
};
```

With the Initializer List, the following steps are followed by compiler:

1. Type's constructor is called first for "a".
2. Parameterised constructor of "Type" class is called to initialize: variable(a). The arguments in the initializer list are used to copy construct "variable" directly.
3. The destructor of "Type" is called for "a" since it goes out of scope.

7.2 Access Control and Encapsulation



In C++ we use **access labels** to enforce encapsulation. A class may contain zero or more access labels:

- Members defined after a **public** label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined after a **private** label are accessible by the member functions of the class, but are not accessible outside of the class. The private sections encapsulate (e.g., hide) the implementation from users of the class.

If the class is defined with the `struct` keyword, then members defined before the first access label are public; if the class is defined using the `class` keyword, then the members are private.

Quick Concept Check: which members are private members in the following?

```
class Fraction {
  private:
    int numer;
    int denom;
  public:
    Fraction(int);
    void print();
  private:
    Fraction();
};
```

```
struct Fraction {
    int numer;
    int denom;
};
```

```
class Fraction {
    int numer;
    int denom;
};
```

We can now redefine `Sales_data` again with proper access control for encapsulation.

```
class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
       bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    double avg_price() const
       { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Quick Guideline: In designing a class, we typically put **data** members into the `private` section and put **member functions** into the `public` section. A typical class definition has the following form:

```
class className
{
private:
    // data member declarations
public:
    // member function prototypes
};
```

7.2.1 Friends

If the data members of `Sales_data` are private, our `read`, `print`, and `add` non-member functions will no longer compile. Why?

A: Although these functions are part of the `Sales_data` interface, they are **not members of the class**. They cannot access the private data members of the class.

A class can allow another class or function to access its nonpublic members by making that class or function a **friend**, with the keyword **friend**:

```
class Sales_data {
// friend declarations for nonmember Sales_data operations added
friend Sales_data add(const Sales_data&, const Sales_data&);
friend std::istream &read(std::istream&, Sales_data&);
friend std::ostream &print(std::ostream&, const Sales_data&);
// other members and access specifiers as before
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&);
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
// declarations for nonmember parts of the Sales_data interface
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
```

Quick Exercise: let us modify the `Circle` class to incorporate `private` and `public` accessible concepts:

```
#include <iostream>
#include "Circle.h"
using namespace std;

int main(){
   Circle c1;
   print (cout, c1);
   c1.setRadius(1);    // This sets c1 radius to 1
```

```
    print (cout, c1);
    Circle c2(2.5);    // This sets c2 radius to 2.5
    print (cout, c2);
    return 0;
}
```

```
Circle radius: 0 area: 0
Circle radius: 1 area: 3.14159
Circle radius: 2.5 area: 19.6349
```

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle{
public:
    Circle() = default;
    Circle(double r){radius = r;}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
private:
    double radius = 0.0;
    const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c){
    os << "Circle radius: " << c.radius << " area: "
       << c.getArea() << endl;
    return os;
}
```

Q: Will the program compile without error?

A:

No! `radius` is a private member and cannot be accessed by non-member functions.

You can fix the problem by (1) adding a simple public utility function `getRadius`, (2) declaring `print` as a friend of class `Circle`, or (3) make it part of your class.

Solution (1)

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle{
public:
    Circle() = default;
    Circle(double r){radius = r;}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
    double getRadius() const {return radius;}
private:
    double radius = 0.0;
    const double PI = 3.14159;
};

std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c){
    os << "Circle radius: " << c.getRadius() << " area: "
       << c.getArea() << endl;
    return os;
}
```

Solution (2)

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>

struct Circle{
public:
    friend std::ostream &print(std::ostream &os, const Circle &c);
    Circle() = default;
    Circle(double r){radius = r;}
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
private:
    double radius = 0.0;
    const double PI = 3.14159;
};
```

```
std::ostream &print(std::ostream &os, const Circle &c);

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &print(ostream &os, const Circle &c){
    os << "Circle radius: " << c.radius << " area: "
       << c.getArea() << endl;
    return os;
}
```

Solution (3)

```
#include <iostream>
#include "Circle.h"

using namespace std;

int main(){
    Circle c1;
     c1.print (cout);
     c1.setRadius(1);
     c1.print (cout);

     Circle c2(2.5);
     c2.print (cout);

     return 0;
}
```

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include <iostream>
// This sets c1 radius to 1
// This sets c2 radius to 2.5

class Circle{

public:
    Circle(double r=0.0):radius{r}{}
    std::ostream &print(std::ostream &os) const;
    void setRadius(double r){radius = r;}
    double getArea() const {return PI*radius*radius;}
```

```
private:
    double radius = 0.0;
    const double PI = 3.14159;
};

#endif // CIRCLE_H
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

ostream &Circle::print(ostream &os) const{
     os << "Circle radius: " << this->radius << " area: "
        << this->getArea() << endl;
     return os;
}
```